CHAPTER 1

Introduction

For physical simulation and a natural, interactive movement of models in virtual environments the detection of collisions is a crucial technology. The general term of collision detection can be split into three parts – *collision detection*, which is the test if two or more objects collide (a decision problem), *collision determination*, which determines which parts of the objects intersect (a computation problem), and *collision response*, which answers the question, which action should be taken in response to a collision (a computation problem). Most work on collision detection considers polygonal models as the least common basis of all surface models. But with large curved models reasonable model parts must be identified in the face set first in order to get efficient approaches. For collision detection and proximity computation in virtual environments, the computation time is paramount over the computation precision.

The goal of this thesis is the design of algorithms for collision detection and proximity computation and its implementation in a scene graph system. In this case we have chosen OpenSG [124], partly due to the fact that we have been involved in the OpenSG PLUS project. Because of the wealth of different model representations, in this work polygonal models and Catmull-Clark subdivision surfaces are covered. Concerning the possible model movements, the rigid-body movements like translation, rotation and scaling, and arbitrary model point movements (deformations) are handled.

Firstly, we give a short overview about scene graph systems for realtime rendering with rasterization hardware, and next we introduce collision detection and proximity problems with their subtle variants. Finally, we state the contribution of this thesis and give an outline of its chapters.

1.1 Scene Graph Systems

Scene graph systems are used to represent general scenes for realtime rendering with rasterization hardware. They free the user from low-level problems, like processing the scene contents in multiple threads (**Threading**) and loading and storing scenes in one of several file formats (**File Formats**, **Textures**). Nearly all systems come with their own file formats in text and binary encoding. The binary encoding is a serialization of all scene objects with type information and handling of platform specialties.

For the definition of renderable units geometric primitives (**Primitives**) have to be associated with render materials (**Material**) and placed in a parent coordinate system. To achieve this, a rooted directed graph structure (DAG, **Structure**) is established, in which a path from the root node defines a renderable unit. Scene data (like geometric primitives and materials) can be shared in different places of the graph, which helps bounding the memory consumption.

Technically, these systems are libraries in object-oriented programming languages and use interfaces to the graphics hardware, e.g., OpenGL [134] or DirectX [38]. With each interface there are several possibilities to transfer geometric primitives to the graphics hardware. For non-changing geometry Display Lists are a retained structure offered by all systems for maximum rendering performance. Different structures are available for changing, indexed geometry like Vertex Arrays and Vertex Buffer Objects (VBO). The following table 1.1 gives an overview of the major systems available today.

Scene Graph	Open Inventor (Coin3D)[1]	Performer 3.1[2]	OpenSG 1.6[3]	OpenSceneGraph[4]	NVSG 1.0[5]	OGRE 1.0[10]
Platforms	SGI Irix, Sun Solaris, Mac OS/X, Linux, MS Windows	SGI Irix, Linux, MS Windows	SGI Irix, HP-UX, Mac OS/X, Linux (32Bit/64Bit), MS Windows (32Bit)	Linux (32Bit), MS Windows (32Bit)	Linux (32Bit), MS Windows (32Bit/64Bit)	Linux, Mac OS/X, MS Windows
Language	C, C++ LGPL (Coin3D)	C, C++ Commercial Binary License	C++ ANSI, LGPL	C++ LGPL	C++ Free Binary License	C++ LGPL
Structure	DAG	DAG	Tree split into Node / NodeCore	DAG distinguishing between inner and leaf nodes	DAG	Tree split into SceneNode / MovableObject
File Formats (for scenes)	IV - Inventor ASCII / Binary, VRML 1.0, VRML 2.0, DXF (Dime), Open Flight (Profit)	Cosmo Binary, Performer Binary, VRML 2.0	Import (OSB, OSG - OpenSG ASCII / Binary, OBJ, VRML 2.0, DXF, 3DS) Export (OSB, OSG, VRML 2.0)	Import (IVE, OSG - OpenScenegraph ASCII / Binary, 3DC, 3DS, AC3, DW, DXF, FLT, Freetype, IV, LOGO, LWO, MD2, OBJ, TXP, DirectX) Export (IVE, OSG)	Import (NBF, NVSG – NVSG Binary / ASCII, NVB, VRML 2.0) Export (NBF, NVSG)	XML mesh and skeleton
Textures (1D, 2D, 3D)	RGB, JPG, TGA, BMP, TIF, GIF, PNG	RGB, JPG, TGA, TIF, GIF, PNG	DDS, JPG, BMP, TIF, RGB, GIF, PNG, DAT (Raw 3D), MNG, BIN, RAW, SGI	BMP, DDS, GIF, MNG, PIC, PNG, PNM, QT, RGB, TGA, TIF, XINE	DDS, JPG, TGA, BMP, TIF, RGB, GIF, PNG (Any Size)	PNG, JPEG, TGA, BMP or DDS files, including unusual formats like 1D textures, volumetric textures, cubemaps and compressed textures (DXT/S3TC)
Manipulators (for primitives)	CenterballManip, TabBoxManip, TrackballManip, HandleBoxManip, TransformBoxManip, JackManip, TransformerManip, (DirectionalLightManip, PointLightManip, SpotLightManip)	None	MoveManipulator, RotateManipulator, ScaleManipulator	None	TrackballTransformManipulator	None
Material (for primitives)	Standard OpenGL	Standard OpenGL	Standard OpenGL, Cg and CgFX Effects, GLSL Vertex and Fragment Program, Per-Pixel Phong Material, Fresnel Material, Multi Texture, Multi Pass	Standard OpenGL, FX Effects, GLSL Vertex and Fragment Program, Multi Texture, Multi Pass	Standard OpenGL, Cg and CgFX Effects, Multi Texture, Multi Pass	Standard, Vertex and Fragment programs, both low-level programs written in assembler, and high-level programs written in Cg, DirectX9 HLSL, GLSL, Multitexture, Multipass, Material LOD
State (technical)	Implicit as traversal state (flags for override and ignore)	State sets (pfGeoState, inheritable)	State sets (non-inheritable, extensible)	State sets (inheritable)	State sets (inheritable)	Unknown
Primitives	Indexed (Display Lists, Vertex Arrays), NURBS Curves, NURBS Surfaces, External: Terrain	Indexed (Display Lists, Vertex Arrays), Text, Ruled surfaces, B-Spline surfaces, COONS surfaces, NURBS, Loop surfaces, Catmull-Clark surfaces[2]	Indexed (Immediate, Display Lists, Vertex Arrays), Text, NURBS, (Loop surfaces, Catmull-Clark surfaces)	Indexed (Display Lists, Vertex Arrays, VBO), Adaptive Terrain	Indexed (Display Lists, Vertex Arrays, VBO)	Indexed (vertex buffers, index buffers, vertex declarations and buffer mappings), Biquadratic Bezier patches, Progressive meshes, Adaptive Terrain
Threading	Locking, Thread-safe render traversals (Coin3D with Version 2.0)	Own Model APP-CULL-DRAW	Very General Model: Multi-Buffered Field Data, High-level synchronization by ChangeList	Own Model UPDATE-CULL-DISPATCH-DRAW	Locking (single writer / multiple readers)	None
Specialties	OpenGL	Frame Rate Control, OpenGL	Regular Volume Rendering, Scene Graph Optimization, Skeletal Animation by Cal3D[11], OpenGL	Paged LOD, Skeletal Animation by Cal3D[11], OpenGL	Support for SLI (PCI/-Express NVidia Clusters), OpenGL	Skeletal Animation, Static geometry batcher, Render System switchable to OpenGL and DirectX

Table 1.1 Side by side comparison of different scene graph systems.

In Chapter 2 we turn to the details of the OpenSG scene graph system used in this thesis. The graph

[1] Open Inventor http://oss.sgi.com/projects/inventor, http://www.coin3d.org

[2] OpenGL Performer http://oss.sgi.com/projects/performer, partly migrated from former OpenGL Optimizer

[3] OpenSG http://www.opensg.org, http://opensg.vrsource.org

[4] OpenSceneGraph http://www.openscenegraph.org

[5] NVSG SDK http://www.nvidia.com/object/nvsg_home.html

[6] Java3D SUN Developer http://java.sun.com/products/java-media/3D/

[7] PLIB http://plib.sf.net

[8] SGL http://sgl.sf.net

[9] OpenRM http://openrm.sf.net

[10] OGRE http://www.ogre3d.org

[11] Cal3D http://osgcal.sourceforge.net

structure with simple bounding volumes for the nodes allow for important rendering optimizations. In most cases just a small amount of Geometry nodes is visible due to their relative location to the camera eye point (**Frustum Culling**) or due to occlusion (**Occlusion Culling**). The exploitation of occlusion to speed up rendering is considered in Chapter 8.

1.2 Collision Detection

Collision detection and proximity computation can be seen as additional services in a scene graph system. In detail these are

Static Collision Determination. Given the current scene graph state, detect if geometry in the scene graph is colliding. Additionally, report the face pair, the points and normals at the collision.

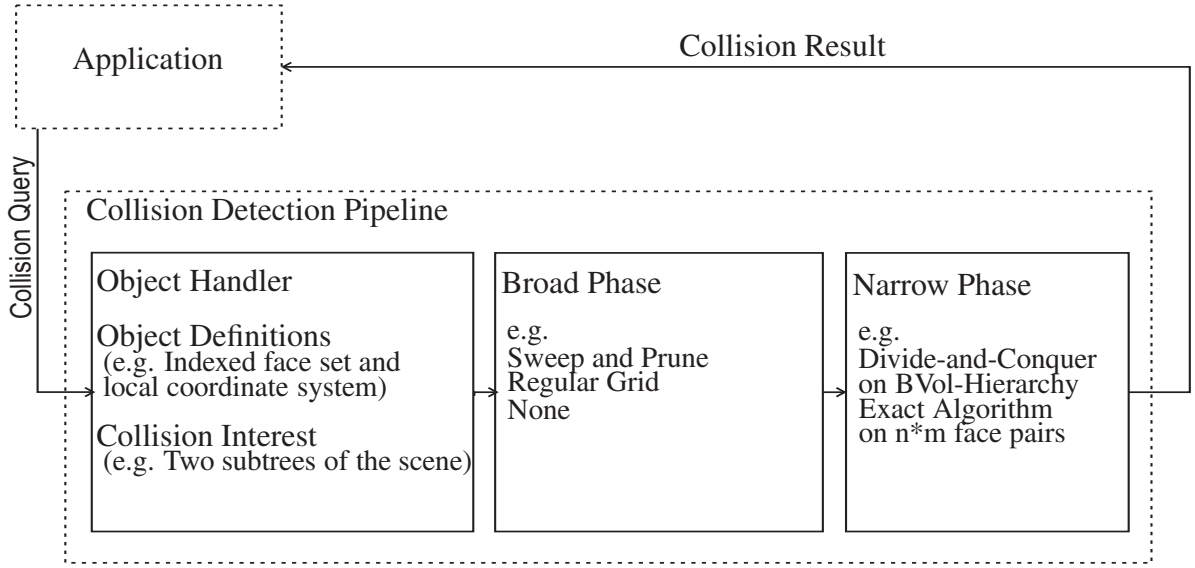
Pseudo-Dynamic Collision Determination. These variants of collision determination address the connection between the collision determination service and the simulator. With pseudo-dynamic collision determination, the simulator performs small step sizes and performs static collision determination for each step.

The pseudo-dynamic collision determination has the advantage that static collision tests are quite fast compared to dynamic collision tests. Potential problems are that the simulator step size has to be small for fast moving objects or there can be issues of one object passing through another in the worst case. Also the simulator should be able to work with penetration cases.

Dynamic Collision Determination, also called Continuous Collision Determination. In dynamic collision determination, the problem is extended by a time parameter. It must be detected if there is a collision in the given time interval between consecutive frames, and the first time of intersection must be reported. Then, the first time and location of collision is known and larger time steps are possible.

Minimum Distance Computation. Given the current scene graph state in which there is no collision, compute the minimum distance between geometry in the scene graph. Additionally, report a face pair and the points having minimum distance.

In general the computation can be structured into a pipeline much like the rendering pipeline. The collision detection pipeline is shown below with its pipeline stages. The front end consists of the object handler, which allows us to define and identify the objects and to state the application's collision interest.



A first neighbor-finding stage is entered, which reduces the set of all objects to smaller neighbor sets of the interesting objects. Within each neighbor set a pairwise algorithm is performed. For this pairwise stage several algorithms have been proposed. For example, hierarchies of simpler bounding volumes on the primitive set are used to stop the search for colliding primitives in sub-quadratic time. We present the realization of this pipeline framework from a software engineering point of view and its integration into the scene graph system in Chapter 7.

1.3 Thesis Contribution and Outline

The most general approach for proximity computation uses hierarchies of simple bounding volumes containing model parts. Several bounding volumes have been proposed including spheres [78], axis-aligned bounding boxes (AABB) [148], oriented bounding boxes (OBB) [96], discrete orientation polytopes (k-DOP) [85] and arbitrary convex hulls [40]. Here, the performance depends on the tightness of the bounding volume, the efficiency of the intersection test for the bounding volume and the strategy for hierarchy generation. The previous work on bounding volume hierarchies is introduced in Chapter 3, Section 3.2.

For k-DOPs a major performance weakness is the intersection test if arbitrary model rotations and scalings are allowed. Therefore, in this thesis a fast table-based algorithm for the k-DOP realignment is developed and presented in Chapter 4. Using the realignment algorithm a pairwise collision detection and minimum distance computation is possible. This contribution was presented in Christoph Fünfzig and Dieter Fellner. Easy Realignment of k-DOP Bounding Volumes. In *Proceedings Graphics Interface*, pages 257–264, Halifax, Canada, June 2003.

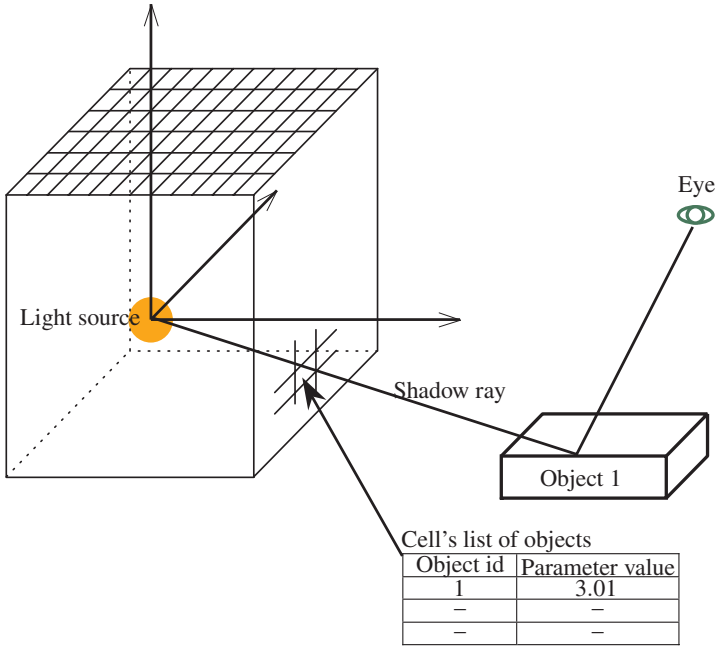


Figure 1.1 Light buffer used to speed up shadow tests during ray tracing [45, 108]. Each cell entry of the light buffer lists the objects intersected by a ray passing through the cell.

Because bounding volume hierarchy approaches require considerable time for update after model deformations, we extended the work with methods using spherical model representations in Chapter 5. These also allow for efficient model rotation and scaling. In Section 5.1 we derive a regular sampling of the sphere corresponding to the six sides of a cube. The resulting hierarchical spherical distance field

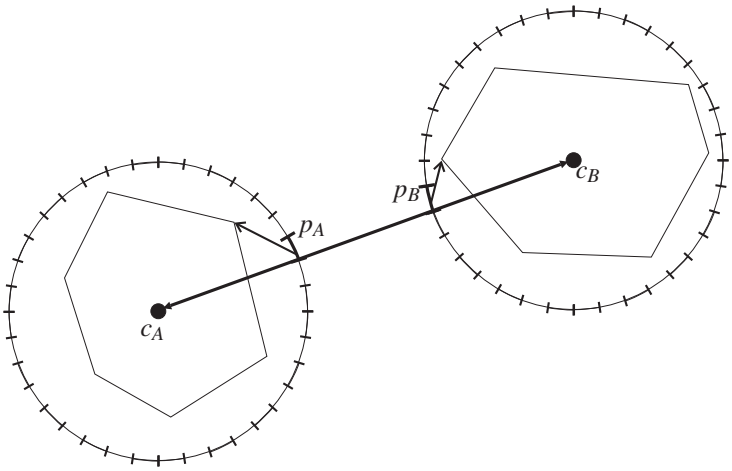


Figure 1.2 Directional look-up table to initialize the intersection test for convex polytopes by Voronoi marching [39].

enables an efficient collision detection with flexible collision information. It can be stored in a space-efficient way without compromising the algorithm's runtime. These results were published in the journal Christoph Fünfzig, Torsten Ullrich, and Dieter Fellner. Hierarchical Spherical Distance Fields for Collision Detection. *IEEE Computer Graphics & Applications*, 26(1), 2006. Spherical representations have not been so popular in the past. They have been used mainly as caches to initialize other precise algorithms. In ray tracing, the spherical cache, called *Light Buffer*, allows us to conservatively find candidates for the shadow tests for point source lighting, as shown in Figure 1.1. For collision detection of convex polytopes a directional lookup has been proposed to initialize the intersection test, called *Voronoi marching*. The table assists in finding a nearby feature to initialize the polytope marching.

Since an update of the regular sampling after model deformations is difficult, the complete recomputation is easier. Section 5.2 applies spherical Bézier surfaces for a more model-dependent approximation. In this thesis a spherical Bézier approximation is used for the first time for quick rejection of large model parts during collision detection between deformable models. A description of how to compute the Bézier approximation for polygonal models and the deformation model similar to freeform deformation [133] follows.

Chapter 6 covers distance computation and collision detection for freeform models in Catmull-Clark representation. Here, a Bézier approximation of the limit surface is proposed, which can be used for numerical distance computation and for collision detection. The Bézier approximation allows a unified handling of regular patches, irregular patches and patches with special features like sharp edges or dart vertices. The collision detection proceeds by using the convex hull of its control points as bounding volume and subdividing the surface patch in case of intersection. For both subproblems, a large body of methods is available, which is presented succinctly in Chapter 3, Section 3.1.

Chapter 8 considers occlusion culling in a scene graph system, where many concepts and algorithms from collision detection can be transfered. The contribution of this chapter is a new organization of the occlusion culling queries into a standard front-to-back-sorted traversal. Optimizations for maximum performance like state sorting and front-to-back sorting are analyzed in detail.

Finally, Chapter 9 gives a detailed summary of this work and concludes with possibilities for future research. Section 9.1 especially addresses the thesis contribution.

1.4 Notation

This section provides an introduction to the notation used in this text.

For points of an affine space we use small letters (p, q, ...). For vectors in its corresponding vector space we use small letters with a right arrow above them ($\vec{v}$, $\vec{w}$, ...). In the coordinate representation of points or vectors we always assume column vectors. We use the symbol $\mathbb{S}$ to denote the unit sphere around a fixed center point c. For points on the sphere and the corresponding radial vectors we sometimes omit the right arrow above them.

We use single bars $|\vec{v}|$ for the lengths of vectors in $\mathbb{R}^n$ measured in the 2-norm $|\vec{v}| := \sqrt{v_1^2 + \ldots + v_n^2}$, if not referring to a different norm in the context or in the subscript explicitly. We denote normalized vectors by $\vec{v^0}$, i.e., $\vec{v^0} := \frac{\vec{v}}{|\vec{v}|}$. For a computation sequence in an algorithm we put superscripts within parentheses to distinguish them from other superscripts, for example $p^{(i)}$, $\vec{s}^{(i)}$. We denote the scalar product between points and vectors by a dot symbol, i.e., $\vec{v} \cdot \vec{w}$; we only pay attention to vector formats if the vector is in coordinate representation, and we use transposition $\vec{v}^T$ to change a column vector into a row vector and vice versa.

For a matrix $A \in \mathbb{R}^{n \times m}$ we use $A_{i,j} \in \mathbb{R}$ to denote the entry in the i-th row, j-th column, we use $A_{i,\cdot} \in \mathbb{R}^m$ as short form for the i-th row, which is a row vector, and $A_{\cdot,j} \in \mathbb{R}^n$ as short form for the j-th column, which is a column vector naturally.

We prefer to write higher derivatives with an operator $\mathrm{d}^{(i)}$ for the i-th derivative and $\mathrm{d}_{\vec{v}}^{(i)}$ for the i-th derivative in direction $\vec{v}$. Partial derivatives are written with the coordinate in the subscript, for example ∂_x, ∂_y, ∂_z.

The Kronecker symbol

$$\delta_{i,j} := \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}$$

is used to write some equations in a short form.

CHAPTER 2

OpenSG

OpenSG is a rendering library for desktop graphics applications as well as multi-screen projection systems like Powerwalls and CAVEs. It has been developed by Dirk Reiners [124] and colleagues since 2002 and extended in the BMB+F (German Ministry of Education and Research) project OpenSG PLUS. It uses a scene graph as its paradigm for scene representation. The next Section 2.1 presents the core system for building the scene graph structure and its functionality. The presentation is based on OpenSG version 1.6 from August 2005. Subsection 2.1.4 gives some technical details about fieldcontainers used as the main data model. The data model contains some specialties for OpenSG's multi threading support (Section 2.2) and for the cluster synchronization (Section 2.3). The chapter concludes with an overview about concepts for association of secondary data structures with the scene graph, which is necessary for applications like collision detection.

2.1 Core System

2.1.1 Scene Graph Structure

Early scene graph systems use a directed acyclic graph (*DAG*) to represent the scene. OpenSG uses a bipartite graph to store the scene. `Node` objects build up a rooted tree, where each instance references a `NodeCore` object, which stores function-dependent data. The wealth of different `NodeCore` classes make up the system functionality, and the following Section 2.1.2 gives some examples.

All `Node` objects have some common attributes. One is the bounding volume. The bounding volume of a node is a simple volume, usually an axis-aligned box or a sphere, that encloses the contents of all the nodes below the current one. It is used by the scene graph to check the node for visibility. If the bounding volume of the node is outside the view frustum (*view frustum culling*), everything below it cannot be visible and does not have to be passed to OpenGL at all. For large scenes this can have a significant impact on rendering speed.

Different scene graph systems have slightly different organizations. OpenSG keeps the list of children in every node, even if it is not used for leaf nodes, since it unifies the structures and simplifies traversals. It also keeps a pointer to the parent node. A traversal visits each node and executes a functor depending on the traversal type and the node type. In OpenSG the traversals `RenderAction` for scene graph rendering and `IntersectAction` for ray intersection are available.

The tree structure of the OpenSG scene graph has the following consequences:

1. Reusing scene parts

   Reuse of a scene part requires the creation of the same tree structure with `Node` objects, while sharing the referenced scene data in `NodeCore` objects (available as function `NodePtr cloneTree (const NodePtr& root)`). For modification of the scene data there is another function `NodePtr deepCloneTree (const NodePtr& root, const std::string& shareString="")`, which clones all `NodeCore` objects of types not contained in the comma-separated type list `shareString` (e.g., `"Transform,MaterialGroup"`).

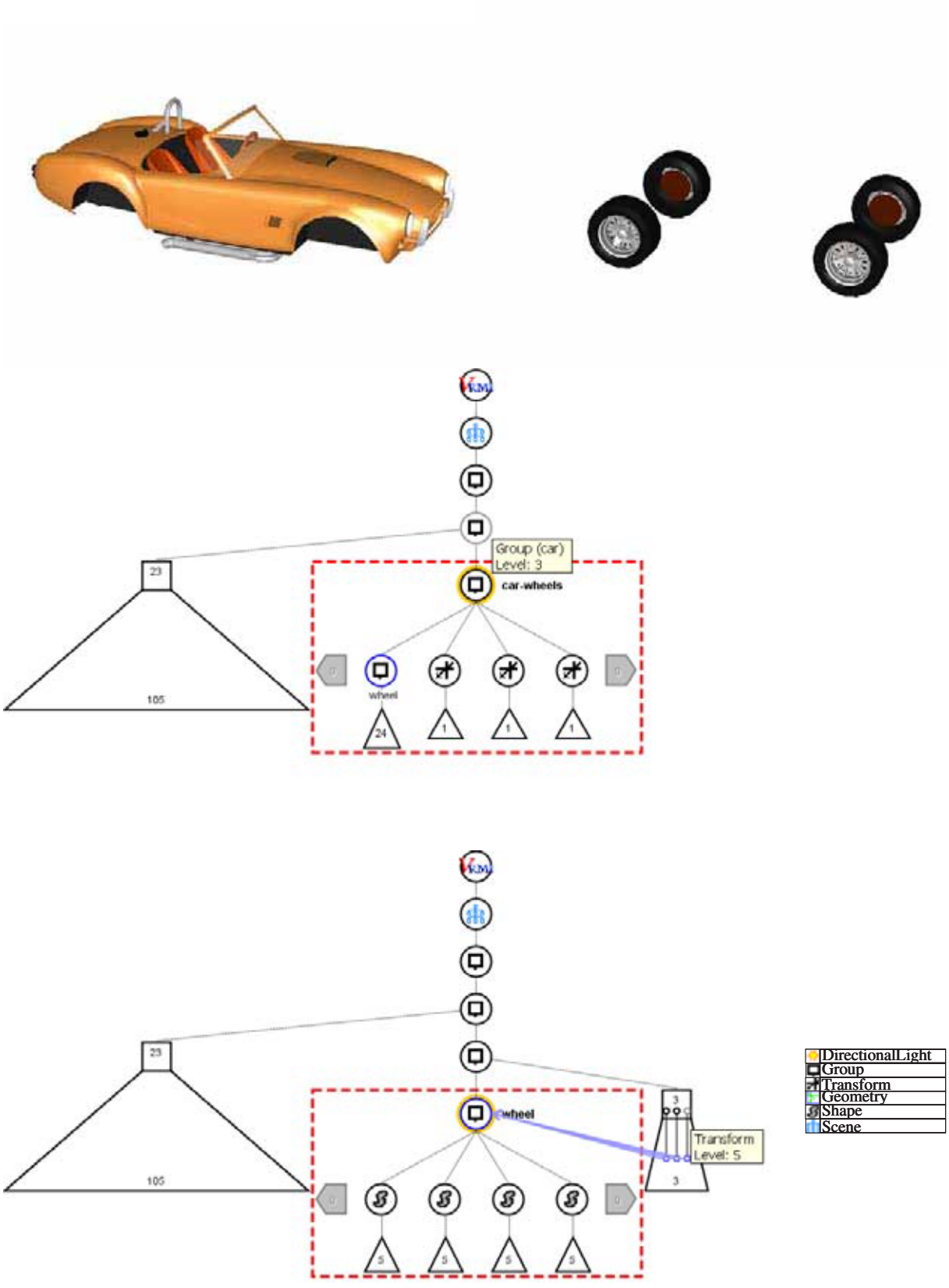


Figure 2.1 Car model with two parts, the car body and its four tires, all instances of a tire prototype (top row). Corresponding scene graph structure (bottom row).

Figure 2.2 Reuse of a scene part (here a single Geometry) with a DAG-based scene graph (left image) and the OpenSG scene graph (right image).

2. Path identification

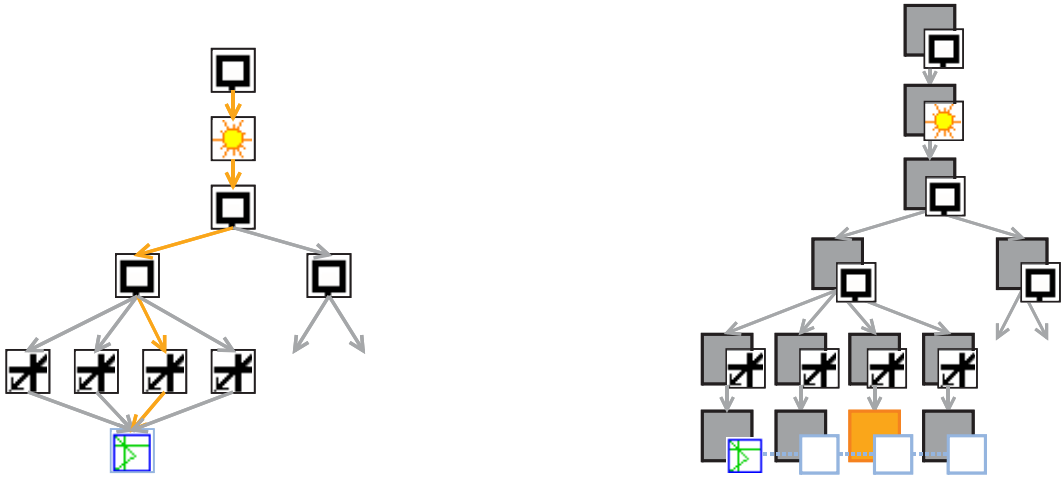


Figure 2.3 Identification of paths by a node sequence in a DAG-based scene graph (left image) and by a single node in the OpenSG scene graph (right image).

Path identification is a problem in DAG-based scene graphs. A node sequence from the root is necessary to uniquely identify a path and the corresponding scene part. With a tree structure like in OpenSG, a reference to a single node also identifies a unique path, namely its path to the root.

2.1.2 Scene Graph Functionality

Each node in the tree structure is assigned a type by its `NodeCore` object. The `NodeCore` objects make up the scene contents and the `NodeCore` objects on a path from the root usually define a renderable scene part. Leaf nodes usually are of type `Geometry`, which contains OpenGL primitives like points, lines, triangles, connected triangles, quadrangles and polygons.

Other node types for inner nodes can roughly be categorized into

1. Groups

   `Group`, `ComponentTransform`, `Transform`, `MaterialGroup`, `InverseTransform`, `Switch`, `DistanceLOD`, `Inline`, `ProxyGroup`, `Billboard`

2. Lights

   `DirectionalLight`, `PointLight`, `SpotLight`

3. Drawables

   `Geometry`, `Particles`, (NURBS)`Surface`, `Slices`, `DVRVolume`, `DynamicSubdivisionCC`, `DynamicSubdivisionLP`